TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

CASPER VRANKEN

COMBINATION OF IOT FRAMEWORK WITH LIQUID SOFTWARE

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 2nd May 2018

ABSTRACT

CASPER VRANKEN: Combination of IoT framework with liquid software
Tampere University of Technology
Master of Science thesis, 43 pages, 1 Appendix page
May 2018
Master's Degree Programme in Engineering Technology
Major: Engineering technology Electronics-ICT
Examiner: Prof. Kari Systä
Keywords: IoT, liquid software, liquidIoT, NodeJS

To mass-deploy and manage IoT applications, an IoT framework was developed by TUT. The capabilities of this framework have been expanded to include liquid functionalities. To limit the extra work an IoT programmer has to add to their IoT applications, the liquid functionalities were added to the application non-specific code rather than the application specific code. To limit power consumption, a polling technique was introduced to check for changes in the state of the applications. To limit the data communication, two ways were created to communicate state changes between applications. One uses a peer-to-peer topology to communicate and the other a master-slave topology. Synchronization collisions are also solved using timestamps.

A network of four IoT devices was used to test the speed of the liquid functionalities as well as the amount of communication between the devices when synchronized. It was found that cloning takes marginally longer than migrating or forking, that liquid transfer speeds are greatly influenced by the presence of a resources folder and that communication between devices works as predicted. To limit power consumption when initiating a liquid transfer, a new way to initiate a liquid transfer has been discussed. It migrates the power to the RR rather than the IoT device. Data communication can be limited by saving all synchronized applications on the device instead of using a syncID.

PREFACE

To conclude the final year of engineering technology at the joint program of UHasselt and KU Leuven, I wanted an experience I would never forget. This is why I chose to go do my thesis on Erasmus in a country I have always appreciated and admired before, Finland. Internet of things has always fascinated me, because of the rapid growth of the amount of devices connected to the internet and the growing power these devices have. I believe this is a field worth investing time in.

This interest in Finland and IoT has led me to the department of pervasive computing at Tampere University of Technology, where a lot of resources are invested in IoT and liquid software. Liquid software was unknown to me, but interested me from the moment I researched it. This led me to do my thesis on both of these topics.

I would like to thank my supervisor at TUT, prof. dr. K. Systä and my supervisor at the joint program of UHasselt and KU Leuven, dr. K. Aerts for providing assistance and valuable feedback regarding the thesis. I would also like to thank M. Sc. F.A. Ghohandizi for providing information about the IoT framework and helping me resolve any issues I encountered with it. Lastly, I would like to thank my parents for giving me the unforgettable opportunity to study and live abroad.

I sincerely hope my work contributed to the field of IoT and liquid software and the future development of the IoT-framework.

Tampere, 23.5.2018

Casper Vranken

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

CD	Continuous Development
DSR	Design Science Research
DOM-tree	Document Object Model tree
IDE	Integrated Development Environment
IoT	Internet of Things
LfD	Liquid.js for DOM
LfP	Liquid.js for Polymer
MVC	Model View Controller
QR Code	Quick Response Code
RR	Resource Registry
TUT	Tampere University of Technology
UI	User Interface
URL	Uniform Resource Locator
WoT	Web of Things

1. INTRODUCTION

Internet of Things (IoT) is the connection between everyday devices, with a certain level of intelligence, through the internet. IoT is becoming more popular by the minute. These devices range from fire detectors to health monitors and from automatic cars to smart homes [1, 2, 3]. Right now, these devices collect the data and send it to an external server that processes the data of all devices. Programming off the devices itself will become possible as these devices get smarter and technology advances. To program these devices, a system with a web browser-based IDE has been proposed and is under development by Tampere Univeristy of Technology (TUT) [4]. This system can be used to develop IoT-applications, mass deploy these applications and monitor the used IoT-devices. More information on how this system is composed and how it functions, can be found in Chapter 3.

Besides this system for IoT, TUT has also been active in the field of liquid software. The number of devices connected to the internet that a person owns, will increase dramatically in the following years [2, 5]. This will eventually lead to liquid software. Liquid software is a concept that states that data, state and applications should be able to move freely between multiple devices and screens [6]. This means, that when writing an email with an email application on a smartphone the email can transfer to a computer with the help of a simple swipe. Then, the user can continue writing his/her email with another email application on the computer, without any disruption. Several structures and frameworks have already been developed for liquid software [7]. Liquid software has also been integrated in vendor-specific applications such as Apple's Handoff and the Google Documents app [8, 9]. There are 4 main use cases in liquid software: migration, forking, cloning and forwarding.

The purpose of this Master's thesis is to combine the IoT application framework developed by TUT and the first three use cases of liquid software. This means that applications running on devices deployed by the system should be liquid. Support for liquidity should also be integrated in the browser based IDE. This expands the possibilities of the IoT system and makes it more versatile. When implementing liquid software, synchronization issues arise. These issues are addressed and possible solutions are proposed.

The following Chapter explains the background and some terms. Chapter 3 is the literature study. This explains the IoT system and analyses the different architectural decisions and use cases for liquid software. In Chapter 4, the method of how the liquid software is combined with the IoT system is explained. Here, the synchronization issues are addressed as well. The results of this combination with a short demonstration of all use cases are presented in Chapter 5 and discussed in Chapter 6. Here, future work is also proposed. Conclusions are given in Chapter 7.

2. BACKGROUND AND TERMS

Liquid software is a vision or concept that states applications and data should not be constricted to one device, but should be able to move freely between devices. As already mentioned in the introduction, there are four main use cases respecting liquid software: forking, migrating, cloning and forwarding. Forking an application is making a copy of the original application and deploying it to the target device, so that both devices run the same application separately. Migrating an application is the same as forking it, except that the original application will be deleted from the source device. This means that only one instance of the application will run at any point in time. An example of forking in everyday-life could be playing music on your phone and then fork the song to every speaker in the house while still continuing to play on your phone. This is illustrated in figure 2.1. The difference with migrating is that the music would stop playing on your phone after the liquid transfer. An example of migration is depicted in figure 2.2. In both of these use cases, the persistent data and state have to be transferred to the source device too. Persistent data or storage is data that is saved in the application for next usage sessions. State is data that is only temporarily saved in the application, but includes the values of the variables in the application. Cloning an application includes forking it, but keeping all application synchronized. An example of cloning an application is writing simultaneously with multiple people on a document. This is depicted in figure 2.3. The final use case is forwarding, where the inputs are taken from one device and forwarded to the application running on another device. Similarly, the outputs can be taken from the device running the application and forwarded to another device [7]. An example of forwarding is typing an e-mail on your phone but using the keyboard for your laptop. This example is illustrated in figure 2.4. Forwarding is not necessary for IoT devices, as inputs and outputs can be accessed with URLs..

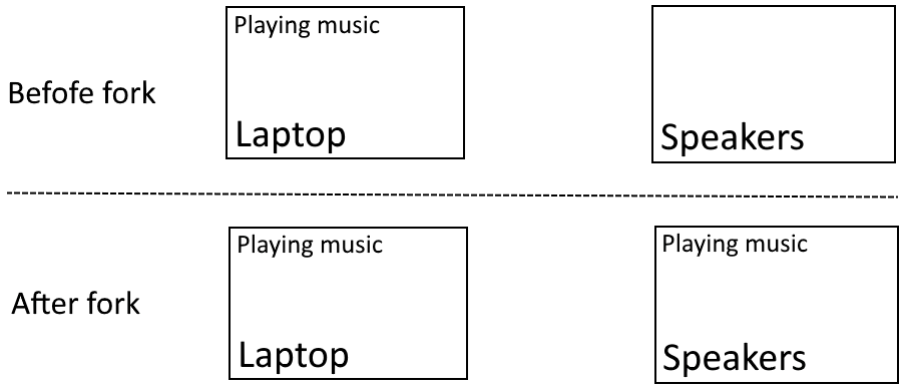

Figure 2.1 *The state of 2 devices before and after forking.*

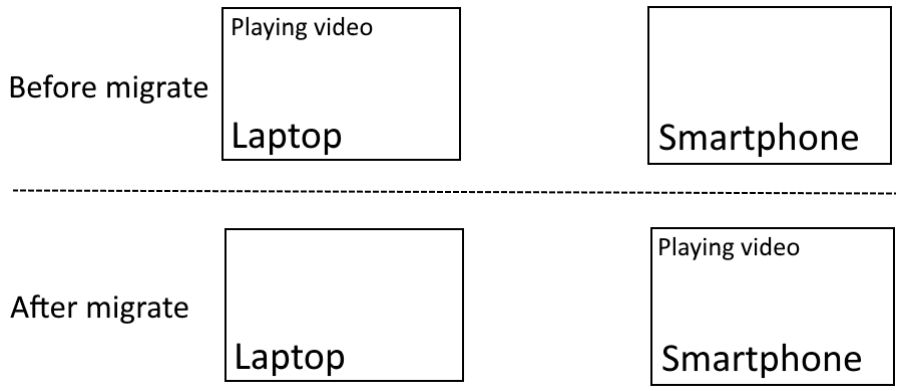

Figure 2.2 *The state of 2 devices before and after migration.*

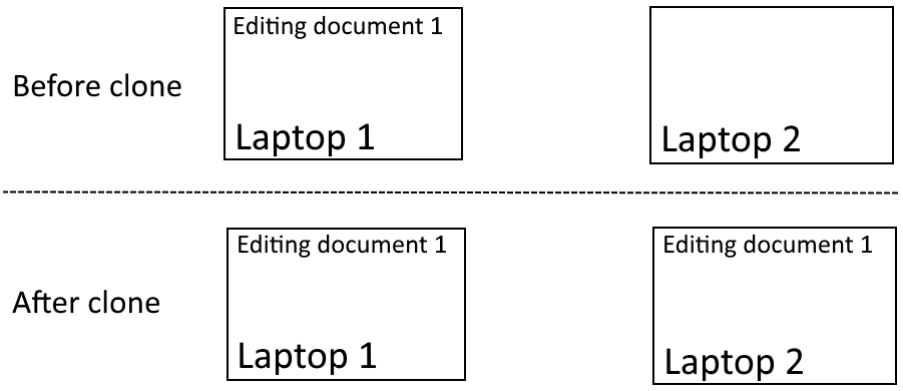

Figure 2.3 *The state of 2 devices before and after cloning.*

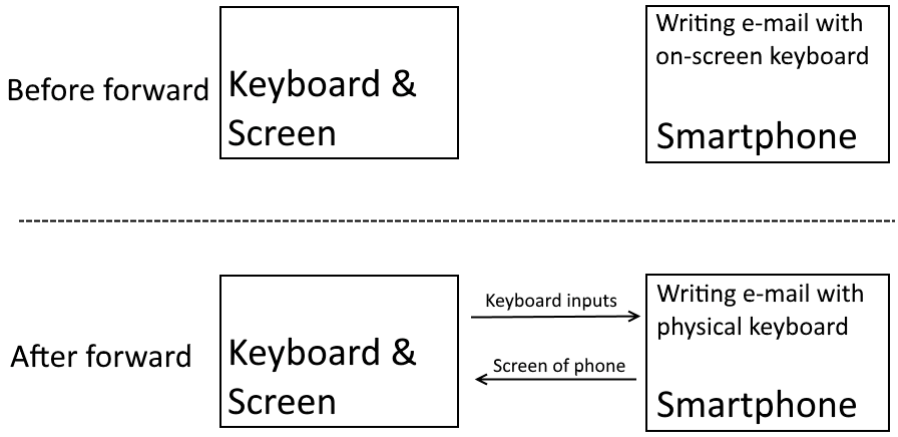

Figure 2.4 *The state of 2 devices before and after forwarding.*

3. LITERATURE STUDY

The literature study will be divided into two parts. The first part will elaborate on liquid software. Here, the key requirements are explained for software to be liquid, followed by the maturity levels of liquidity for applications. Then, the architectural design space will be explained. Finally, two existing liquid frameworks will be compared. The second part of the literature study will explain the IoT framework. The structure is thoroughly explained and all separate components are elaborated on.

3.1 Liquid software

3.1.1 Key Requirements

According to [5], there are six key requirements for a casual multiple device ownership world. In this Chapter, they will be elaborated on.

The first requirement states that users should be able to switch between all the devices connected to the Internet that they have and continue the usage as before. This is one of the basic concepts of liquid software and is essentially what liquid software stands for. An effect of this is that all the devices available to the user, should be known to the software. This can be achieved through a process called discovery. Discovery of devices can be done through a multitude of ways such as QR codes, Wi-Fi and Bluetooth [7]. QR codes hide the complexity of long URLs and remove the tedious process of typing a long link.

The second requirement states that changing between devices should be as easy and casual as possible. This can be done multiple ways. Some of which include URLs, QR codes and Wi-Fi that consequently trigger the liquid transfer. This QR code or URL will be generated by the device that requests a liquid transfer and can be scanned by the device to which to transfer. If a device is known through Wi-Fi discovery, the relative location of the host and target device can be calculated. However not trivial to implement, this can enable geometrical based gestures for transferring the liquid application [7, 10, 11]. Maintenance and management of the device should be hidden from the end-user to improve user friendliness.

The fourth requirement is that when a transfer occurs, the full state of each application shall be transferred or recreatable, so that the user can continue their activities on another device. Web-browsers use Document Object Model-trees (DOM-trees) to represent data and state. Transfer of state can be achieved through the use of virtual DOM-trees. DOM-trees consist from a root <html> tag and then a <head> and <body> tag, which on their turn consist of other nodes [12]. In this approach, only the initial virtual DOM tree is loaded from the server. When the user requests a liquid transfer, only the differences between the initial DOM-tree and the current DOM-tree are sent to the receiving device. This way it suffices to send only the deltas instead of the entire state, which greatly reduces the amount of data that needs to be sent [13].

The fifth requirement states that roaming between devices should not be tied to a single vendor ecosystem. Nowadays, most liquid applications are tied to a single vendor. This should not be the case in truly liquid applications. Examples of vendor based liquid applications are Handoff by Apple and Google docs [14].

The sixth and final requirement states that the user should be in full control about the liquidity of his or her applications and data. For privacy reasons, the user can choose to not save certain data on certain devices. When migrating or cloning to another user's device or a public device, strict control policies should be defined to ensure privacy and security of data [7].

Today, automatic synchronization is still an exception rather than the norm. According to [5, 7], automatic synchronization will become the norm for data and applications.

3.1.2 Maturity Levels and Layering

According to [15], liquid web applications can be evaluated with maturity levels. This can be done by dividing applications into layers, according to the Model View Controller (MVC) pattern, and defining a maturity model regarding the liquidity of websites ranging from solid applications to liquid web applications. The model is based on three facets, which will each be elaborated on. First logic deployment will be discussed, then data and state storage and finally the communication channel will be discussed. Each facet can be divided into three levels.

Logic deployment (Controller)

The first layer is the controller layer. This is where the web application executes its tasks. The three levels regard the client thickness. The first level is an ultra-thin client. Here, the only entity that can do logic is the server. No logic can be done on the client-side, meaning scripting languages such as JavaScript can not be used. The only logic present on the client concerns the retrieving and displaying of the data. The second level is a thin client. Here the logic is shared between the server and the client. The server can offload part of its logic to the client. Views can be altered by user input and made responsive. The third and final level is a thick or rich client. At this level, the logic is entirely deployed on the client-side. These thick client can also be aware of other thick clients connected to the web. This is very helpful for liquid applications, as the view can adapt depending on the other devices. For a thick client, enough computing power should be available on the client side. Off-line operation becomes more accessible for thick clients [7].

State storage (Model)

The second layer is the model layer. This is where the persistent data of the application is stored. The three levels regard where the data is stored, ranging from a server system to a distributed system.

The first level is a centralized storage system. Here, all data-management is deployed on the server-side. This is good for consistency, all devices of one user can have the data as the data is stored on the server. However, privacy suffers from this approach if no proper security is provided. As said in Chapter 3.1.1, the final requirement for liquid applications is that the user should be in full control of the liquidity of their data. This is obviously not the case in this approach. All data is stored on the server, the user cannot decide where to store their data. Examples are centralized MySQL [16] and NoSQL [17] databases. No off-line mode is possible for the clients.

The second level is decentralized storage system. Here, data is stored both on the server- and client-side. Cookies are a form of a decentralized storage system by caching the data received from the server. The client can also choose to use the client-side as a primary data storage system, and the server-side as a backup. This method of data storage enhances data privacy. This is only the case when a direct connection between devices can be set up. It also enhances performance during off-line operation, as long as the data is cached beforehand. Another benefit is that less data should be downloaded from the server, meaning that a weaker Internet

connection is possible.

The third and final level is a distributed system. Here, the data is stored solely on the client-side and none on the server-side. This enhances privacy as the data is only stored on the users devices, unless the devices are not properly secured. Data is shared between devices through a peer-to-peer channel. It is, however, more challenging to maintain data synchronization across all devices as data is stored in multiple places. Another problem is the likeliness of devices being off-line for prolonged periods of time, not being able to synchronize data. In this approach, only a local internet connection is necessary [7].

Communication channel (View)

The third layer regards the communication channel on which data is exchanged. The three levels concern the direction of communication and whether the clients can communicate between each other or not. Any higher levels contain the functionalities of the previous level. For example, level 3 contains the features from level 1 and 2.

The first level is a client-server pull. Here, only the client can pull data from the server. An example of this is RESTful HTTP. Cloning and forwarding of the application state is not obvious with this approach, but can be achieved by regularly polling the state of the server. The second level is a client-server push. Here, a duplex connection is opened by the client to the server. An example of this are Web-sockets. Here, cloning and forwarding is a possibility. The third level is a peer-to-peer communication channel. Here, clients communicate with each other without the need of a server. This can reduce latency, as the number of hops can be decreased. This can, for example, be achieved by the WebRTC protocol.

Maturity levels

With the three facets of the model divided into three levels, maturity stages can be defined based on these levels. The first stage are Web 1.0 applications. These applications only have the view on the client side and the rest on the server side. The applications use the first level of all facets.

The second maturity stage are rich web applications. Rich web applications use a thin or ultra-thin client meaning that the client can execute logic. It is possible for rich web applications to use a decentralized storage system. Rich web applications are still on the first level of the communication channel facet.

The third maturity stage are real-time web applications. Real-time web applications can use centralized and decentralized storage systems, but are still restricted to a thin client. Real-time web applications can however use the second level of the communication channel facet. These applications can not only pull data from the server, but also push data. All liquid web applications should be at least at this maturity level.

The fourth stage are hybrid web applications. These applications use the same level as real-time web applications for the model and controller facet, but are at the third level for the communication channel facet. Communication is done in a peer-to-peer fashion. Latency is thus significantly lower.

The fifth and final maturity stage are peer-to-peer web applications. These applications are at the third level of every facet. They use a distributed storage model, a rich client and peer-to-peer communication.

3.1.3 Architecture

In this section, several architectural considerations will be elaborated on. They contain the key elements to form the technical choices or design space discussed in section 3.1.4. Four architectural considerations are proposed by [7].

A major element of liquid software is the ability to benefit from the advantages of each device. A truly liquid application should consider different input and output methods for different devices. A smaller screen should probably display the data in a different way than a larger screen. Here, UI adaptation plays a major role.

Another element is data and state synchronization. In software, there are two kinds of application data, persistent and dynamic. Nowadays, persistent application data is stored locally on each device and can be synchronized using cloud-based solution. These are, unfortunately, often vendor based. Liquid software includes not only synchronization of persistent data, but also dynamic application data. An example of dynamic application data is the zoom-level or position of a web-page. These dynamic application data can be caught at different levels of granularity. Identification is needed to apply the correct user settings to the correct user.

The third architectural consideration is the client/server partitioning. This has been widely discussed already in section 3.1.2. The final element is data security. This has also been touched on by section 3.1.2. People using liquid software should be able to decide the liquidity of their data.

3.1.4 Design space

This section discusses some of the technical choices that need to be made to develop a liquid application. This is also often called the design space. The design space is formed from the architectural choices discussed in Chapter 3.1.3. Following technical choices are proposed by [7] and [18]. Each element of the design space is discussed in a separate section.

The topology of a liquid architecture can be centralized, decentralized or distributed. This has already been discussed in Chapter 3.1.2. This topology has the same structure for state replication as well as application source distribution. The topology of the state replication and application source distribution can differ within one application.

Discovery is a major part of liquid software as the device that wants to do a liquid transfer must know to which device it can transfer. The process of finding these devices is called discovery and can be achieved in a number of ways. The first approach is by creating a personal network with either Wi-Fi or Bluetooth and then connecting devices. Another approach is creating a server and linking devices through shared URL- or QR-codes. Both of these approaches fall under the term existence discovery. The third method of discovering new devices is location discovery. Through certain methods, relative positions can be calculated through Wi-Fi or Bluetooth. To migrate or clone a session, specific geometrically based gestures can be used (for example swiping left or forward). The final approach is ownership discovery. This is based on the authorization of users. When a user is authorized on a certain device, it can initiate a liquid transfer to it. Authorization can be achieved in a number of ways, such as user/password combination, smartcards and shared secrets.

Layering of liquid software is already discussed in Chapter 3.1.2 with the MVC-model. The choice between a thick and thin client can be made based on a number of criteria. These criteria include computing power, energy consumption and required bandwidth.

The following paragraphs discuss the part of the software that is responsible for the liquid transfers. This ranges from operating system level to component level, each having its own benefits.

The lowest level is the operating system level. If liquid software is implemented at this level of granularity, liquid software would not be tied to certain applications, but all applications would be liquid by default. It is the most complete but also most

complex way of implementing liquid software. Problems include security issues, hardware differences and resource consumption. Another struggle would be that every device should run the same operating system.

At a higher level, virtual machines and containers can be used. The transfer of running all applications can be achieved and it is the most adopted system for doing so. Problems with limited bandwidth, can be solved by selecting parts of the virtual machine or container that need to be transferred.

The next level is the application level. It is probably the most natural way when thinking of moving a running application. In the previous method, all applications would have liquid capabilities by default. Here, all applications would have to be programmed to be liquid.

The final level is the component level. Here, only parts of the application will be transferred. This positively impacts bandwidth usage as less data should be sent to the target device. Every component has to be programmed to be liquid.

There are two major alternatives to deploy applications to a client device. The first alternative is on-demand applications. Here, no installation is required and the application is downloaded when necessary. The second alternative is pre-installed applications. On-demand applications can only be used in on-line mode, except if the application is cached. Pre-installed applications can always be used in off-line mode in a non liquid manner.

Liquid software deals with two types of data. The first is persistent user data. This kind of data needs to be stored so that it is available to all devices of the user. The second type is the application state. This data must be stored in a manner so that it is easy to transfer the data when a liquid transfer is requested. During cloning and forking, conflicts may occur because multiple devices are working on one application. These conflicts need to be solved at the application level. State synchronization can happen in two major ways, as briefly discussed in Chapter 3.1.1. The first possibility is trickle updates. This way, all changes are updated as soon as they are made. The second possibility is batch updates. Here, changes are cumulated and updated after a period of time. The latter approach is useful when updating devices that have been off-line for prolonged periods of time. The last consideration to be made when choosing how to update the state and data, is the way the data gets to the device. The first possibility is pushing. Here the device that is sending initiates the communication between devices. The second possibility is pulling. In this approach, the device that is receives data initiates the communication between devices.

3.1.5 Comparison of the liquid software programming frameworks

In this final section of the literature study regarding liquid software, two programming frameworks mentioned in [7] will be compared. They are called Liquid.js for DOM (LfD) and Liquid.js for Polymer (LfP). The comparison is based on the design space discussed in Chapter 3.1.4 and described in [7]. These are two frameworks with the same goal, but developed in parallel by different teams.

Overview

On the one hand, LfD is a framework based on virtualized DOM trees. Virtualized DOM trees are a way of quickly manipulating DOM trees through an abstraction layer. It is deployed as a JavaScript file and runs on the client. This means it has to be included in the application. Some initialization code also has to be implemented. On the other hand, LfP is a framework based on web-components and the Polymer project by Google. A LiquidBehaviour class can be instantiated into the definition of a component and can define which properties of the component should be liquid.

Topology and Code Deployment

LfD can use both a centralized topology and a decentralized one. It is made such that any communication protocol can be used to transfer the application. Right now, it uses WebSockets and all data needs to go through a centralized server. LfP however, aims to be as decentralized as possible. With LfP, the states are transfered through a peer-to-peer channel using WebRTC.

Both LfD and Lfp work on an on-demand basis and can be cached afterwards. Individual components can be cached separately when using LfP. With LfP, any client can request any other client their copy of the application to enable client repository paradigm. This is not possible for LfD.

Granularity

For LfD, the entire application is always sent to synchronize. This means that the user can not choose what should be liquid and what not. However, only differences in the DOM tree are sent to limit data consumption. LfP is component based and

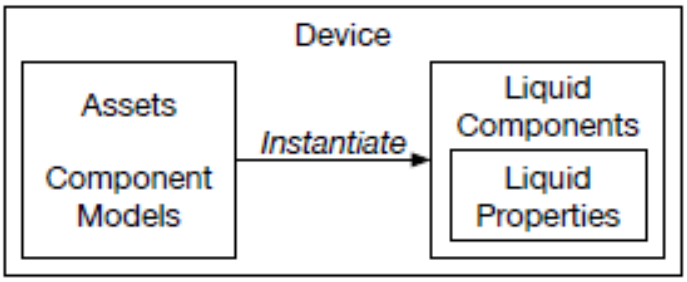

Figure 3.1 *The four levels of abstractions LfP can transfer to another device.*

allows single components to be synchronized independently. This way, it is possible for the user to keep certain parts of the application private.

Liquid User Experience

LfD and LfP support all liquid use cases except forwarding. LfD is developed with migration being its top priority, but both forking and cloning are possible too. For LfP, four levels of abstraction were developed. Devices, assets, components and properties can all be separately transferred to another device. The differences between these 4 levels of abstractions are depicted in figure 3.1 [7]. The devices consist of assets and components. Assets are the definitions of components and components have properties that define how it behaves [19].

State and data

LfD sends the state and data of the application to another device by comparing the initial DOM tree with the current DOM tree and the difference between the two is sent. The developer can choose when to send the state and data. (For example with the press of a button or when a change in the tree occurs). Sequential use is perfect for this, but simultaneous use is also possible because conflicts are minimized. The framework supports both trickle and batch updates [13].

LfP stores the state and data of the application in the clients by default. The developer can however choose to store the state and data in a centralized web-server. Only trickle updates can be used.

Summary

Table 3.1 illustrates a quick summary of all the differences and similarities between the two frameworks discussed in this section.

	LfD	Lfp
Technology	DOM-trees	Polymer
Topology	Centralized	Decentralized
Code Deployment	On-demand with caching	On-demand with caching and client repositories
Caching	No individual components	Individual components
Granularity	Application level	Component Level
Liquid use cases	Migration, forking, cloning	Migration, forking, cloning
Updates	Trickle & Batch	Trickle

Table 3.1 *The differences between LfD and LfP.*

As can be seen, both frameworks have their advantages and disadvantages. However, it can be seen that LfP has progressed further in the liquid field. With its support for client repositories and decentralized topology, LfP enters the hybrid web applications maturity level whereas LfD remains at the real-time web applications maturity level. On top of this, LfP provides more freedom to choose which part of the application is liquid and which part is not. This freedom is not present in LfD. Both LfD and LfP provide the same liquid functionalities. LfD does provide both trickle and batch updates where LfP only provides trickle updates.

3.2 IoT Application Framework

Nowadays, IoT devices have relatively little computing power and only retrieve data and send it to the cloud. But according to [20, 21], IoT devices will become more powerful and will eventually be able to host applications. Following is a close analysis to the dynamic, distributed platform, named liquidIoT, developed by TUT and the requirements it needs to satisfy. Lastly, continuous development (CD) for IoT programming is discussed.

3.2.1 Implementing distributed systems

The first alternative that needs to be considered is the communication protocol between peers, the two obvious alternatives being Web Services like SOAP and WSDL and the RESTful style. SOAP, being resource-hungry and complex, is complicated

for programmers in the usage of these services. REST provides scalability and is straightforward. IoT devices very often provide simple interfaces and functionalities, so REST is the obvious choice between the alternatives. Using REST as an interface for IoT applications is a concept called Web of Things (WoT). In the platform suggested in [21], the user interacts directly with the peers through their URL rather than with a mash-up that interacts with the peers and filters the data. A service discovery mechanism needs to present because users often need to interact with multiple peers. This mechanism has to have a group communication mechanism in place for bulk operations.

Only services are not enough to make a distributed system. In IoT, innovation is often a requirement for finding the best business cases. This requires rapid and regular updates to the devices. Because in IoT not all devices are equally powerful and differ from each other, they do not need the same version at all times. The deployment mechanism should automate the discovery process to avoid human errors when deploying new applications. The deployment mechanism should also be scalable, as many applications need to be deployed and updated with single operations. Both of these reasons call for CD.

3.2.2 LiquidIoT

With the requirements listed above, TUT developed a programming platform for IoT devices called LiquidIoT. LiquidIoT consists of 3 components: the application framework, the runtime environment and the resource registry. All will be explained in detail now including the workflow of deploying an application and the communication between the components.

The application framework

The application framework provides the developer with certain functionalities that he/she needs to fill in with application specific code. There are three functions that need to be filled in, as can be seen in figure 3.2. The first one is *task*. This is the function that gets called on regular intervals. It is also possible to only execute this task once. The second function is *initialize*. This gets called before the task function is called, like establishing a connection. The third and final function is the *terminate* function. This gets called before the applications stops and is used for gracefully killing connections with other peers. Applications can also provide REST interfaces called application interfaces (API) that can be called from anywhere in the network. Figure 3.2 displays all functionalities the application needs.

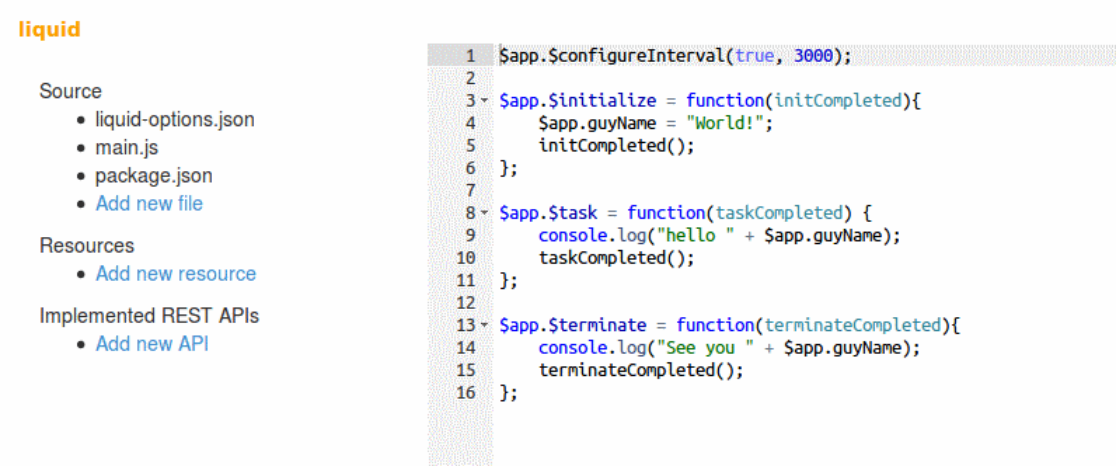

Figure 3.2 *The developer uses this framework to give the application the functionalities it needs.*

After all functionalities are filled in and the API's are in place, the tool packs the application in to one package. This contains a *main.js* file that defines the source code and main liquidIoT functions, which can be expanded by adding other source files, and a *Package.json* file that includes some metadata like name and version. It also includes a *liquidiot.json* file used for discovery and bookkeeping and a folder called resource that contains any images or sounds. The package can then be deployed. The deployment window can be seen in figure 3.3. The table on the right displays all applications that can be deployed on the devices, displayed on the left.

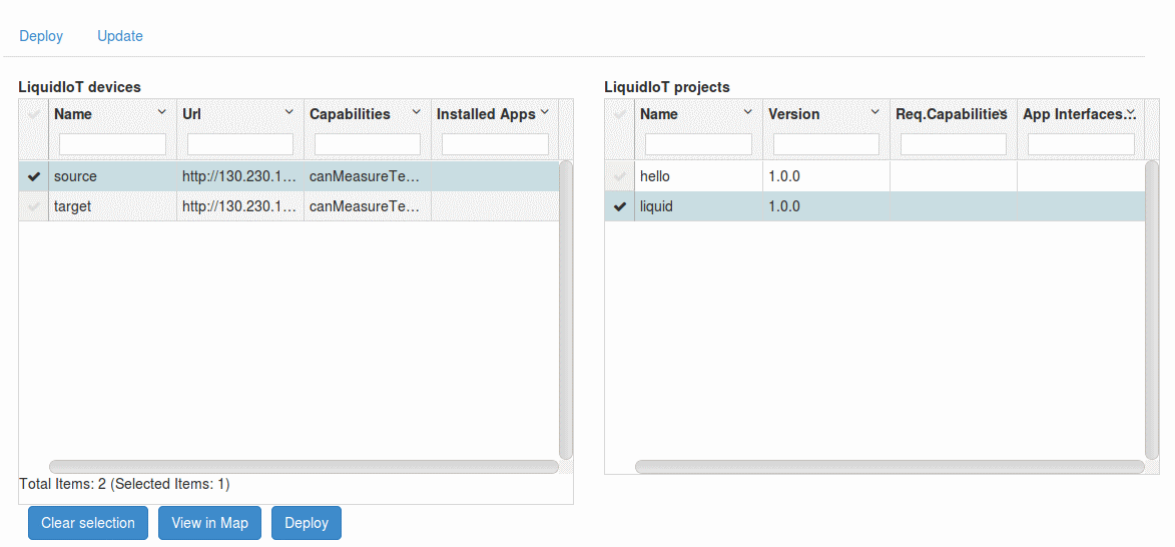

Figure 3.3 *This screen is used to deploy applications to devices. Another tab can be used to manage the applications.*

The runtime environment

The runtime environment is a system based on NodeJS [22] and it transforms any IoT device to an application server that can host IoT applications. The runtime environment receives the code from the framework, prepares the application and notifies the resource registry about its status.

Resource registry

All IoT devices and applications are in this system are managed by centralized system called the resource registry (RR). It also contains a resource discovery system, which will be explained in Chapter 3.2.2. It keeps track of all resources and their capabilities. Resources are divided into four categories: applications, devices, device capabilities and application interfaces.

Application deployment flow

When an application needs to be developed and deployed, the following steps are followed. First, when a new device is installed, it registers itself to the RR. Secondly, the new application needs to be developed or an already existing application needs to be found. Then, all suitable devices are found through the RR. The fourth step is deploying the application through the application management API provided by the runtime environment. Then, the deployment returns the status of the deployment to the development tool and if successful to the RR too. Now the developer can query the application and finally manage and monitor the application through the RR. A diagram is shown in figure 3.4.

Discovery Mechanism

In this Chapter, the technical details of the discovery mechanism used in LiquidIoT are explained. All resources known to the RR are described in JSON format. All devices have an id, name, location and a list of capabilities among other pieces of information. It also has a list of applications which on its turn have an id, name, list of interfaces and many more.

For the RR to find appropriate resources, a query language has to be used. It should be easy to understand for all people that come into contact with it and it should

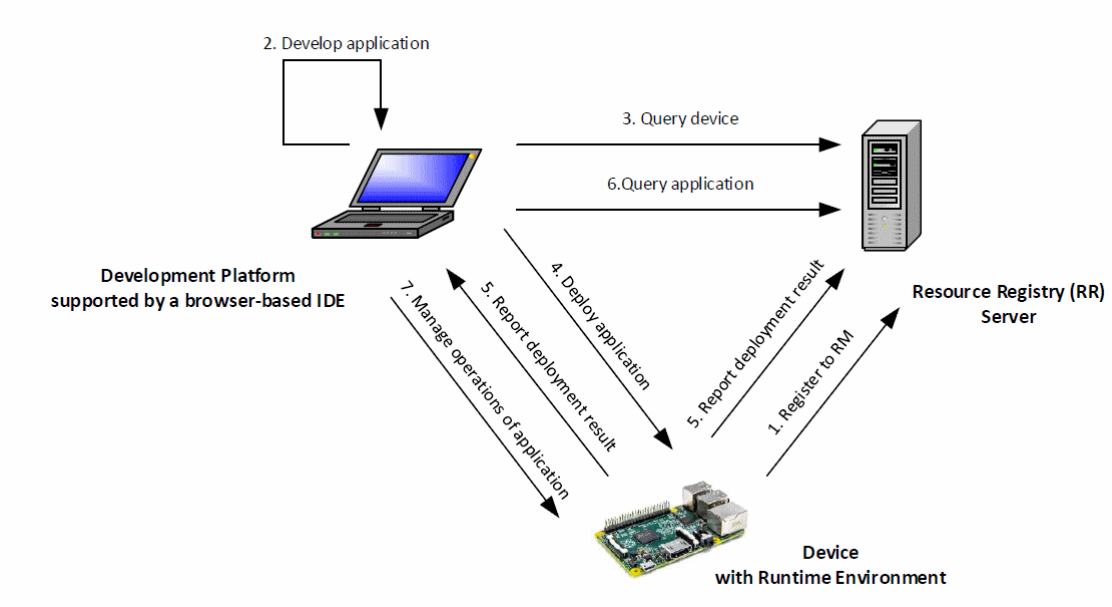

Figure 3.4 *The flow of deploying and managing an application with LiquidIoT.*

support complex query patterns. Currently the ArangoDB Query Language (AQL) is used due to its similarity to the Structured Query Language (SQL) and its support for document oriented structures. On top of this, AQL also has some procedural elements like a for-loop.

3.3 Continuous Delivery

Continuous delivery (CD) is the act of automatically testing of new pieces of code and preparing them for deployment. When a new piece of code is written, it goes through testing. If the code passes all automated tests, it goes to the automated release process. Only the actual release is done manually. This has several benefits such as an accelerated time to market, improved productivity and reliable releases [23].

CD is mostly used in cloud-based infrastructures. However, [20] has found similarities between CD for cloud-based Internet services and IoT applications. When deploying a new application with CD, there are several deployment patterns, all with advantages and disadvantages. The most straightforward method is in-place deployment. Here, the new version replaces the old one and only a short downtime is present. If only the application changes, downtime can be short. However if a change of execution environment is necessary, downtimes may be larger. This method is easy to transfer to IoT.

Another method is rolling deployment. The in-place deployment pattern changes all

applications at once, where the rolling deployment pattern updates the applications sequentially. This strategy takes more time but has the advantage of having zero down-time, as there is always one host running. Another advantage is that if the new version is faulty, the old version can be rolled back before all applications have been changed to the new faulty one. This pattern requires a balance-loader to regulate which application needs to be updated at what time. This is not easily integrable to IoT because of the need of a balance-loader.

One more method that will be discussed is the blue-green deployment pattern. Here, when the application needs to be updated, the old application remains on the device during transition. When the new software is stable, the URL's are swapped and the new version is active. Using this approach, zero down-time is present and a rollback is easy as it only requires changing the URL's back. A disadvantage is that the transfer of persistent data between versions is complicated. With the mindset of IoT devices becoming more powerful, a limited version of the blue-green deployment pattern is integrable.

Staging can also be used with the blue-green deployment pattern. Staging makes use of a staging environment that is an exact replica of the production environment. IoT devices are always tightly connected with the physical world, so a staging area is impractical. However, a simulator can be used. The staging environment can be hosted in the target device as the new application. When the simulation runs correctly, the URL's can be swapped.

Canaries are a final optimization to application deployment. Canaries are a subset of the IoT-devices that host the new applications, while all other devices still have the old application. When all goes correctly with the canaries, it is to be expected that the application will work too in the other devices. Staging, the blue-green deployment pattern with the integration of canaries, seems to be the most stable version of application deployment.

3.4 Conclusion

To develop IoT-applications, a framework has been developed to easily code, manage and deploy these applications. This framework consists of three main components: the resource registry, the IDE and the runtime environment. The IDE is used to create, deploy and manage all applications and devices. The RR is used to keep track of all applications and devices. Finally, the runtime environment is used to run all applications on the devices.

Liquid software is a vision to make all applications liquid, meaning they can seamlessly flow from one device to another. There are four main use cases regarding liquid software. The first is forking. This means making a copy of an application, its state and its storage and deploying it on another device. The second use case is migrating. Migrating is the same as forking except that the original application gets deleted from the source device after the transfer. The third use case is cloning. This is forking an application and keeping state updates synchronized with each other. This means that all cloned applications are always in the same state. The final use case is forwarding, where the inputs and outputs of one device are forwarded to another device.

The design space is an important part to design liquid software. It discusses how the liquid software is implemented. This includes state replication topology, application source topology, the discovery mechanism, layering, granularity, client deployment and state and data. Two frameworks have been developed that implement the facets of the design space differently. LfP is at a higher maturity level and provides more freedom regarding liquid support.

4. METHOD

The goal of the thesis is to implement basic liquid functionalities such as migration, forking and cloning in the IoT framework so that the IoT developer has minimal extra work to implement these liquid functionalities. To achieve this, an incremental and iterative way of working was used [24]. This means that only one problem was being solved at a time. First, the existing code of the IoT framework was analyzed so that further development would go fluently. Then work continued in small steps. The first step in coding added support for communicating between IoT devices. Then, an extra tab was added to the IDE. After every step, testing was done to make sure everything worked as expected. When a test failed, a new design was made, developed and tested again. Each iteration of code was reviewed by the research team led by Professor K. Systä and feedback was provided to improve the code and liquid support. An illustration of the iterative development method can be seen in figure 4.1.

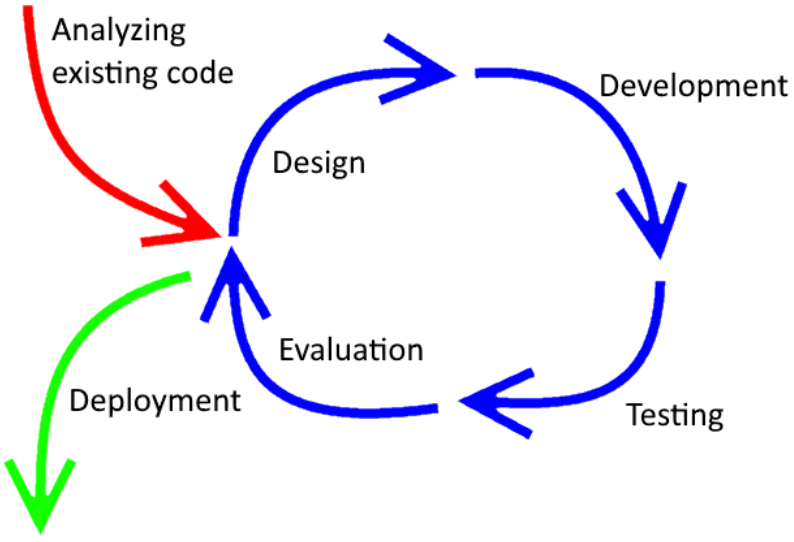

Figure 4.1 *To develop the project, an iterative way of working was used.*

4.1 Design science research

Design science research (DSR) is an important research paradigm for information systems (IS). DSR for IS describes how to properly design a scientific artifact and then research it. An artifact can be defined as multiple things, but in the scope of this thesis, an artifact is a system design [25]. According to [26], there are seven research guidelines for a research to be a DSR. Firstly, the research must provide an artifact of some kind. Secondly, the artifact should solve existing problems. Thirdly, the artifact must be tested. Fourthly, the research must provide contributions to the field of the artifact. Fifthly, the construction and testing of the artifact should rely on tested methods. Sixthly, the artifact should make use of existing and tested methods provided by others. And finally, a DSR must be presented so its clear for technology-oriented as well as business-oriented people.

This thesis provides a solution to implement liquid functionalities to IoT devices within an IoT framework. The design of the functionalities has been described in the following sections of this Chapter. The liquid functionalities are tested and the results of tests and evaluations are presented in Chapter 5. The results are discussed in Chapter 6. The construction of this research was based on a incremental design process, as explained in the previous section. Several packages made by and tested by others were used in the creation of the system, such as the tar-pack [27] package and Lodash [28] provided by npm.

4.2 Migration and Forking

The implementation of migration and forking forms the basis of the other use cases of liquid software. The only difference between migration and forking, is that the application gets deleted from the source device after migration whereas it stays active after forking of the source application. Because of this, both methods run the same code but with an extra function that deletes the application for migration.

4.2.1 Algorithm of migration and forking

For migration and forking, the entire application code, state and local storage has to be transferred to a new device. As can be seen in figure 4.2, the transfer consists of three steps. Firstly, the IDE needs to send a signal to the source device that it needs to migrate or fork its application. This signal contains the applicationID of the application that needs to transferred, the URL of the target device and if

the transfer method should be a migrate or a fork. The sending of the signal does not necessarily have to be done by the IDE. If the signal is correct, the application will transfer regardless of the origin of the signal. Secondly, the source device needs to do the migration or forking. To do this, the source device collects all necessary code-files from its application and stores them in a separate directory. Then it polls the state of the application and saves it in a JSON-format that is then saved in the same directory as the code. Afterwards, the storage gets copied into that same directory. The storage contains all resources like sound- and image-files. Finally, the directory that contains all information is packed into a tarball with the .tgz extension-format. To do the packing, the device uses the tar-pack package [27] provided by npm. This tarball is then sent to the target device that handles it. The handling is explained in the next paragraph. Everything necessary for migrating and forking, from collecting all required files to sending the tarball, is done by the runtime environment. No extra code has to be added to the applications written by the developer. It is important to notice that the IDE also sends a tarball to the target device when deploying a normal non-liquid application, without a state-file. This tarball can also contain a number of resources.

Finally, when a target device receives a tarball, it unpacks it like it would unpack any normal application that it would receive from the IDE. It saves the file that contains the state of the source application for later use. When the application is deployed, a function gets called from the application code before any task is ran. This function reads the contents of the file that contains the state of the source application, and changes its state accordingly. The application then reports its status to the RR and the IDE for user feedback about the transfer. After this, the application code is ran as usual, with the correct initial state.

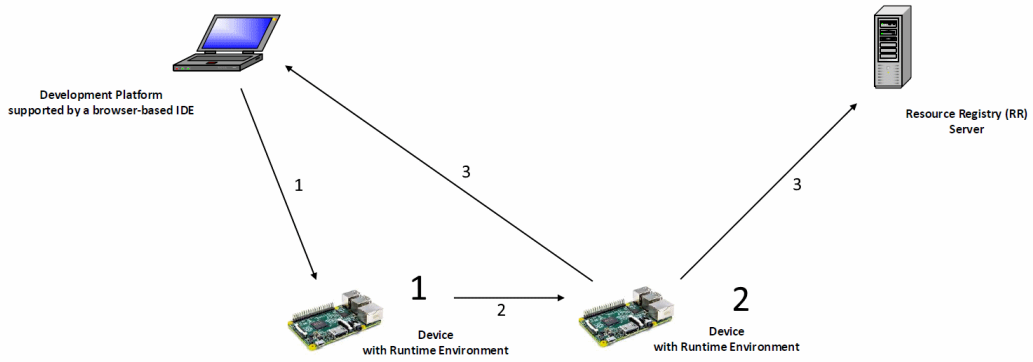

Figure 4.2 *The source device packs a tarball that it can send to the target device.*

4.3 Cloning

To implement cloning, two alternative methods were developed for the communication of state-changes. The first method uses peer-to-peer communication between applications, the second method communicates in a master-slave fashion. Here, the master is the RR and the slaves are the applications. For both methods, a pushing technique is used rather than a pulling technique. This means that when the state of an application changes, it will forward this to other devices rather than the devices asking to other devices if they have had any state-changes. Both methods use the same approach for the transfer itself, as this is similar to forking with subtle additions. The methods for communicating state changes will be explained afterwards.

To keep applications synchronized, each application has a syncID. All applications that are mutually synchronized have the same syncID. Applications that are not synchronized with any other application, have a value of -1 for the syncID. This syncID is saved in a separate file in the application that is used for all liquid purposes. This file right now only contains the syncID. The file is generated in the browser-based IDE. When an application gets deployed from the IDE, the syncID is always -1.

Applications are synchronized if the state of the applications are the same. Synchronization is necessary when a state change happens. A state change is defined as the addition of a variable, the deletion of a variable or the change in contents of a variable within the memory. Applications can be ordered to not synchronize their state and only accept state changes. This way, the application will listen to the applications that are sending out their state changes.

State changes can be detected in three major ways. The first way is by polling the state on a regular basis. This can be done using JavaScript's *setInterval* method. For this approach, the application's state has to be cached in memory and this cache can be compared to the actual state of the application. When a change is detected between the cache and the actual state, synchronization is necessary. The cache is always updated to the actual state when a state has been detected.

The second way is by triggering an event when the state changes. Because JavaScript does not call any event when a variable changes, this is harder to implement. A possible solution can be made by altering the source code of JavaScript itself. The final way is by only sending state changes when a specific function is called, implemented by the developer. This requires extra code for the application developer but expands the liberty that the developer has. This has already been implemented in

LfD [7].

4.3.1 Transfer of applications

When a transfer has been initiated by the user on the IDE, the application that is being cloned first needs to check its syncID. If the value of the syncID is not -1, the application that needs to be cloned is already synchronized with other applications. In this case, the source application can be forked. The syncID, application and state are transferred to the new device. The application on the new device is automatically synchronized with any other applications with the same syncID.

If, however, the value is equal to -1, the application needs to initialize the synchronize operation. It does this by requesting a syncID from the RR. The RR then generates a new random unique syncID and sends it back to the application that needs to be cloned. The application saves this value and a normal fork can be initiated.

4.3.2 Peer-to-peer synchronization of the state

The first method for cloning applications uses a peer-to-peer communication protocol. This means that in principle, no external server is necessary. This method does still require the RR that is present in the IoT framework. It uses the RR for receiving a syncID and requesting synchronized devices. For the communication of the synchronization, 4 steps are defined for the peer-to-peer approach, as can be seen in figure 4.3. The steps are as follows:

1. A state change happens in the first application.
2. Application 1 requests all applications with the same syncID as it from the RR.
3. The RR returns all applications with the syncID that was provided.
4. The first application publishes all state changes to the applications it received.

The RR uses an AQL query to find all applicationIDs and the URLs for the devices and sends it back when requested. The application that is sending the state updates then sends data in JSON format to the second device via a POST method that contains the applications ID (aid), the variables that have been added or changed (data), the variables that have been deleted (dels), the synchronization ID (syncID)

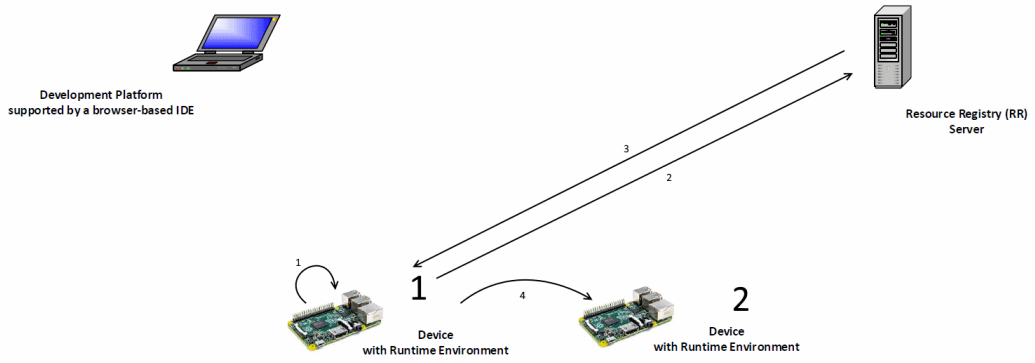

Figure 4.3 *Synchronizing two applications with peer-to-peer communication.*

and the time of the update (time). The device receives the POST data and sends it to the application that changes its state accordingly. An example of the POST-data in JSON format can be seen in figure 4.4.

```
{
    aid: 48948,
    time: 1523448178,
    syncID: 22,
    data: {
        people: [{name: "name1", age: 23},{name: "name2", age: 21}],
        counter: 55
    },
    dels: {}
}
```

Figure 4.4 *Example of data that needs to be synchronized.*

When an application requests all synchronized applications from the RR and it receives an empty array, the device sets its syncID to -1 locally and requests the RR to update its value to -1 as well. After this, the application is not synchronized anymore.

4.3.3 Master-slave synchronization of the state

An alternative method for the synchronization of cloned application uses a master-slave topology. As there is already an external server, the RR, present in the IoT framework, no extra hardware is necessary. It is however possible to create a dedicated server just for synchronization. As depicted in figure 4.5, three steps are needed for this method. The steps are as follows:

1. A state change happens in the source application.
2. The device sends the state change to the external server.
3. The external server handles the state change and forwards it to all synchronized devices.

The external server uses the same method as the option for synchronization to find all synchronized devices and to send the synchronization data to target devices. However, in this method, the external server can decide whether to accept or drop any synchronization requests. This enables more control over the system, as all states are centralized and conflicts can be handled in a centralized place.

Similarly to the first method for synchronization, when a device requests synchronization and no other devices are synchronized with it, the external server will urge the device to change its syncID value to -1 so that it is no longer requesting synchronization.

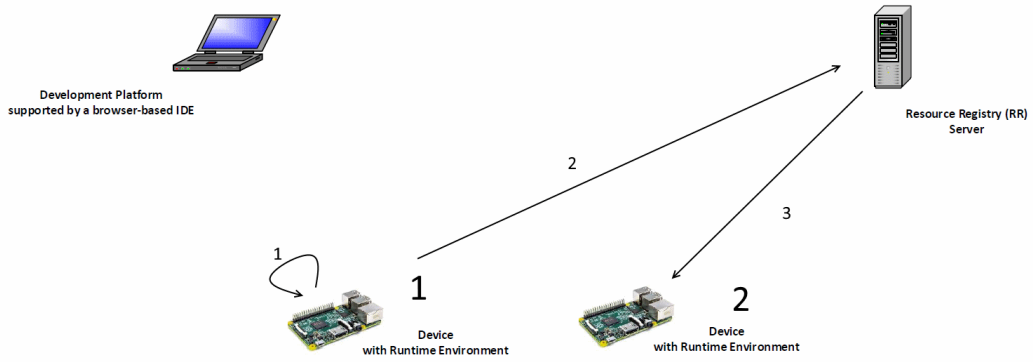

Figure 4.5 *Synchronizing two applications in a master-slave fashion.*

4.3.4 Synchronization collisions

When a state change happens in two applications at approximately the same time, they will send the state changes to each other, which will result in a collisions where the devices have different states. To solve this, [29] has proposed a solution that uses timestamps. Firstly, an example of a certain use case is given which results in a non-synchronized state. For simplicity, time-steps of 1ms are presumed and sending the new state to other applications takes 2ms to travel over the communication channel.

Time	Application 1	Application 2	Communication channel
0ms	State changes to 1	Idle in state 0	Idle
1ms	Idle in state 1	State Changes to 2	Transmitting state 1 to application 2
2ms	Idle in state 1	Idle in state 2	Transmitting state 1 to application 2 and state 2 to application 1
3ms	Idle in state 1	Receive state of application 1	Transmitting state 2 to application 1
4ms	Receive state of application 2	Idle in state 1	Idle
5ms	Idle in state 2	Idle in state 1	Idle

Table 4.1 *When two applications have a state-change close to each other, synchronization issues arise.*

As depicted in table 4.1, two applications can result in different states after a synchronization attempt. This is because a state change happened in application 2 while the state change of application 1 was still transmitting, which resulted in a swap of states rather than a synchronization of states. To solve this, timestamps are added to each state change and each application saves the timestamp of the last state change. When an application receives a state change, it will first check if the state change is newer than the last state change. If it is newer, the state change is accepted, otherwise it is discarded. Only timestamps of accepted state changes are saved. Table 4.2 depicts the same situation as table 4.1, but with the solution in place. The text in red depicts a discarded received state change. The variable t is the saved timestamp of the last accepted state change for the two applications.

Time	Application 1	Application 2	Communication channel
0ms	State changes to 1 (t=0)	Idle in state 0 (t=0)	Idle
1ms	Idle in state 1 (t=0)	State Changes to 2 (t=1)	Transmitting state 1 to application 2
2ms	Idle in state 1 (t=0)	Idle in state 2 (t=1)	Transmitting state 1 to application 2 and state 2 to application 1
3ms	Idle in state 1 (t=0)	Receive state of application 1	Transmitting state 2 to application 1
4ms	Receive state of application 2	Idle in state 2 (t=1)	Idle
5ms	Idle in state 2 (t=1)	Idle in state 2 (t=1)	Idle

Table 4.2 *Synchronization issue solved using timestamps.*

The end state of both application is now the same, with minimal extra resources needed. This is because application 2 discarded the state of application 1. It did this because application 2 itself had already done a more recent state change.

4.4 Technical changes

In this section, the technical changes made in the code for every part of the IoT framework are explained. The code is available on GitHub, the links for the different components of the IoT framework can be found at the end of this section.

In the non-application specific code of the runtime environment, three new endpoints were added on the device level. The first endpoint is to migrate or fork an application running on the device. The second one is to clone an application and the final endpoint is to pass along synchronization data to the correct application. These endpoints then execute the relevant parts of the code to do the liquid transfer. An organized table of all new endpoints at the device level in the runtime environment can be seen in table 4.3.

On the application level, four new endpoints were added. The first endpoint is used to receive synchronization data passed on by the device. The second endpoint is used to save the current state of the application. The third returns the syncID of the application and the final endpoint saves a newly received syncID. Furthermore, a *setInterval* method was added to check for state changes in the application. This method can call request the synchronized devices from the RR and send synchronization data to other applications. A schematic overview of the new APIs added to the application level of the runtime environment can be found in table 4.4. The new endpoints of the device and application are illustrated in figure 4.6.

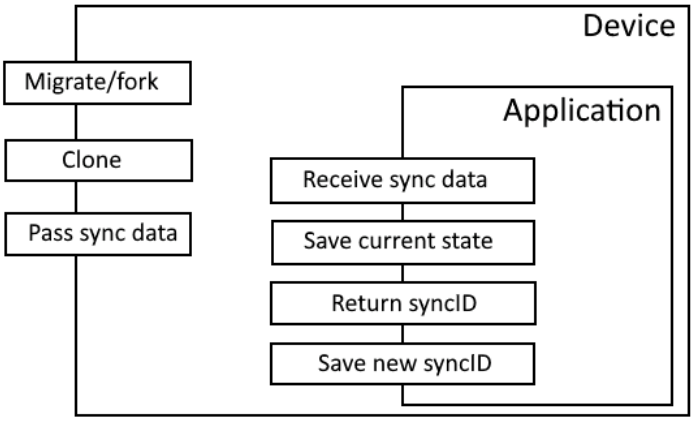

Figure 4.6 *The endpoints added to the device and application level of the non-application specific code.*

The IDE works with Angular 1 and Jade templates. Because of this, no extra design had to be done to implement the extra tab. The use of Jade and Angular 1 resulted in good reusable and reliable code so that little code had to be added to implement the liquid functionalities.

The RR has only been expanded by two new endpoints. The first endpoint is used to generate a new syncID when requested by an application. The second endpoint transmits the synchronization data when making use of the master-slave fashion with cloned applications. These functions make use of queries written in AQL to make sure that no two identical syncIDs are generated and to find synchronized devices. A structural representation of all new APIs for the RR can be found at table 4.5.

4.4.1 REST APIs

This section illustrates all new REST APIs added to the program in a constructive manner.

URL	Method	Parameters	Description
/transfer	POST	id url del	Migrates or forks the application with its id equal to *id* to *url*. *del* indicates if the current application should be deleted.
/clone	POST	id url	Clones the application with its id equal to *id* to *url*.
/sync	POST	aid	Relay the entire body to the application with *aid*

Table 4.3 *The REST APIs present in the runtime environment at the device level.*

URL	Method	Parameters	Description
/sync	POST	time data dels	Sets all applications variables to the ones represented in *data* and deletes the ones represented in *dels*.
/savestate	GET	/	Saves the state to a file.
/syncId	GET	/	Returns the syncID.
/saveSyncId	POST	syncId	Saves a new syncId.

Table 4.4 *The REST APIs present in the runtime environment at the application level.*

URL	Method	Parameters	Description
/generateSyncId	GET	/	Returns a new syncID.
/stateupdate	POST	syncId aid	Sends the body to all applications with the same *syncId* except for the application with id *aid*.

Table 4.5 *The REST APIs present in the resource registry.*

4.4.2 Links to code

The code for the IoT framework components can be found at following links.

Resource Registry: https://github.com/caspervranken/liquidiot-server/tree/CasperRR

Runtime environment: https://github.com/farshadahmadi/liquidiot-server/tree/Casper-clone-p2p

IDE: https://github.com/caspervranken/liquidiot-IDE

5. RESULTS

The three main liquid functionalities, migrating, cloning and forking were added to the framework. In this Chapter, it is explained how the user can use these functionalities within the framework. The functionalities are also tested on speed and correctness. These results are then discussed in Chapter 6.

To include all liquid functionalities in the IDE, a new tab was generated in the deploy and update window. The contents of the new tab can be seen in figure 5.1. On the left hand side, the different applications can be seen that are installed on devices. The table on the right displays all available devices. Three buttons are available for the different integrated liquid use cases: migration, forking and cloning.

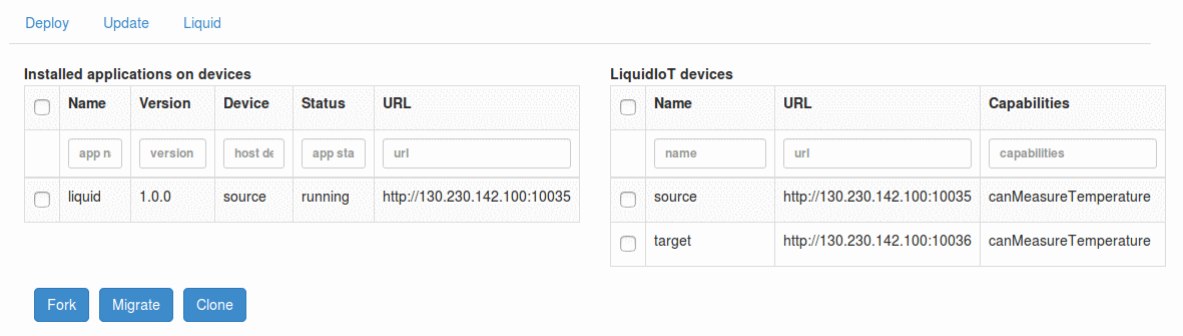

Figure 5.1 *The tab created for all liquid use cases.*

N applications installed on devices can be selected and M devices can be selected on the right hand side to initiate a liquid transfer. Every application will do the liquid transfer to all devices selected. If a liquid transfer is initiated, the IDE sends the list of selected devices, together with the selected liquid use case, to all applications that then handle the request.

To initiate a liquid transfer of any kind, the developer does not need to add any additional code to the application. All applications deployed by the IDE are liquid by default. This is because all liquid functionalities are implemented in the runtime environment and non-application specific code. Because of this, it is not possible to declare components of the application non-liquid.

5.1 Test setup

A network of 4 Raspberry Pi 3 Model B's was installed and connected through a local network to test the altered IoT framework. This enabled P2P communication between the devices. The RR and IDE ran on a Ubuntu virtual machine running on a Intel-i5 dual core laptop with 8GB of memory with Windows 10. This laptop was also in the same network. The database linked to the RR ran on a virtual machine outside the local network. A diagram of the setup is depicted in figure 5.2.

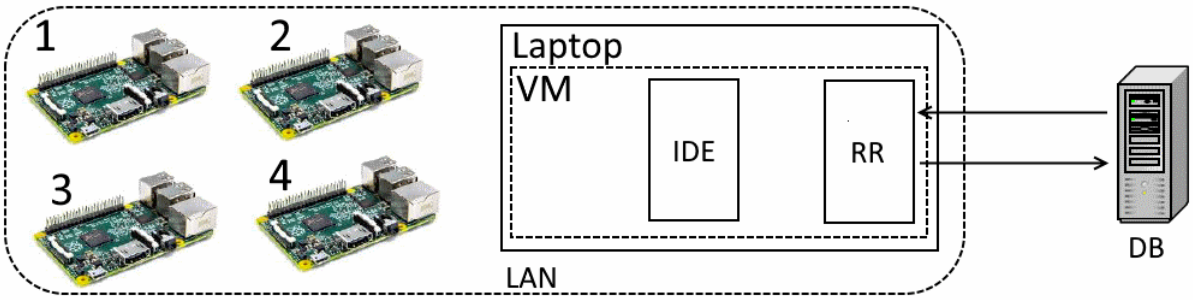

Figure 5.2 *A schematic representation of the test setup.*

To test migration and forking, the time taken from receiving the signal from the IDE to preform a liquid transfer to another device receiving and reporting a good transfer is measured. This time is also dependent on the network characteristics. The same is done for cloning, but the amount of synchronization messages between all devices is also measured. The task of the IoT applications gets called every second. The code depicted in the listings below, is used to test all liquid functionalities of the IoT framework. It contains JSON objects, arrays and normal variables. Measuring the time it takes to do a liquid transfer and the amount of synchronization messages with the same program across the devices, results in a good representation of the traffic on the communication channel and the speed at which the liquid transfers happen.

```
$app.$configureInterval(true, 1000);

$app.$initialize = function(initCompleted){
    $app.people = [{"name":"Name 1","age":21},{"name":"Name 2","age
        ":19}];
    $app.counter = 0;
    initCompleted();
};

$app.$task = function(taskCompleted) {
    $app.counter = $app.counter + 1;
    for(var i = 0; i < $app.people.length; i++){
        if($app.counter%3===0){
```

```
            $app.people[i].age = $app.people[i].age+1;
        }
        console.log("Person " + i + "'s name is " + $app.people[i].
            name + " and is aged " + $app.people[i].age + ".");
    }
    taskCompleted();
};

$app.$terminate = function(terminateCompleted){
    terminateCompleted();
};
```

Program 5.1 *Code for testing the liquid capabilities of the IoT framework*

5.2 Migration and forking test results

Migration and forking were tested by initiating a liquid transfer with varying amounts of resources attached. The amount of resources attached were 0MB (no resources folder present), 1.9MB of resources and 3.6MB of resources. These resources were arbitrary pictures and sound files. The average time needed to do the liquid transfer with a certain amount of resources is given in table 5.1. Another test was used to measure the importance of target devices in migration or forking time. The average time to migrate or fork an application with no resources folder for different amounts of target devices can be seen in table 5.2. The averages were made over five measurements and all raw measurements can be found in appendix A.

Resources (MB)	Time to migrate or fork (s)
0	0.313
1.9	1.597
3.6	1.604

Table 5.1 *The time to migrate or fork the application with varying amounts of resources.*

Amount of target devices	Time to migrate or fork (s)
1	0.313
2	0.306
3	0.316

Table 5.2 *The time to migrate or fork the application with varying amounts of target devices.*

5.3 Cloning test results

Testing for cloning was divided in to two parts, one for the liquid transfer and one for state synchronization. The testing for liquid transfers is identical to the testing for migration and forking. To test synchronization, the amount of synchronization messages was measured during a time span of 10 seconds with a varying amount of devices synchronized. The amount of time needed to clone one application with varying amounts of resources is given in table 5.3. For this test, only one target device was present. The amount of synchronization messages for a certain amount of synchronized devices is given in table 5.4.

Resources (MB)	Time to clone (s)
0	0.494
1.9	1.801
3.6	1.803

Table 5.3 *The time to clone the application with varying amounts of resources.*

Synchronized devices	Amount of messages in 10s.
2	21
3	30
4	42

Table 5.4 *The amount of synchronization messages sent between a certain amount of synchronized devices.*

6. DISCUSSION

6.1 Migration and forking

In the implemented method for migration and forking, the application source topology uses client repositories. Applications are received through the other peers in the network when a sequential transfer happens. The discovery and layering are already provided by the IoT framework. The liquidity operates at the virtual machine level, all applications are liquid by default. This is because the packing is done on the device rather than on the application. The applications are downloaded and installed on the devices, this means they are able to run in off-line mode. The applications are thick-clients, as the model, view and controller are all present on the device running the applications.

As stated in the previous Chapter in tables 5.1 and 5.2, the speed of migration or forking solely depends on the amount of resources present in the application to be transferred. When no resources are present, the average time to migrate or fork is 0.313 seconds. The time to migrate or fork barely changes when doubling the amount of resources. This indicates that including a resource folder adds between 1.2 and 1.3 seconds to the migration or forking. The amount of target devices has marginal impact on the time to migrate or clone.

6.2 Alternative method for migration or forking

The implemented method for migrating and forking uses the power of the devices to pack the state and application of the source device into a tarball and to send this tarball to the target device. An alternative method can be used when the computational power is limited. Here, the IDE sends a request for a liquid transfer to the source device. The source device then saves the state to a state-file and sends it back to the IDE. Because the IDE is used to initially deploy applications, they have the source-code saved for every application that is deployed. It is also possible to save all applications on the RR and use the RR to do the following process. It then packs this application together with the received state-file into a tarball and

sends it to the target device. Figure 6.1 depicts this process. The steps are as follows:

1. A request for a liquid transfer is sent to the source device.
2. The source device polls the state and sends it to the IDE.
3. The IDE packs the application together with the state-file into a tarball and sends it to the target device.
4. The target device reports the state of the deployment.

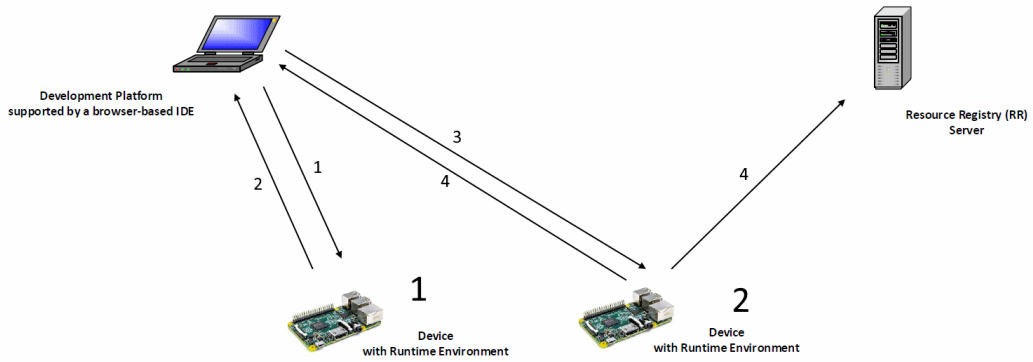

Figure 6.1 *The IDE packs the tarball and sends it to the target device.*

An advantage of using this approach is that the computational power is focused on the PC or laptop running the IDE or the server running the RR rather than on the weaker IoT device. This advantage will however become more irrelevant over time as IoT devices get more powerful over time [5]. A major disadvantage is that the IDE or RR must have every application stored. This is not necessary for the implemented method, as the IDE only needs to have the application stored at deployment. Right now, the RR never stores the applications. The energy consumption due to communication does not play a major role, as the energy cost for setting up a WiFi connection is high relative to the energy cost per bit sent [30] and the number of connections created is equal compared to the implemented method. The design space is similar to the implemented method except for the application source topology, which uses a master repository instead of client-repositories.

6.3 Cloning

To communicate state-changes between applications, there are two alternatives: peer-to-peer and master-slave communication. When using the peer-to-peer alternative, fewer hops are needed for communicating the state-changes. There also is less load on the RR. Both alternatives use a pushing technique rather than a pulling technique. This is because of the size of the network of the devices. The framework is made to support thousands of IoT devices, each running the synchronized application. When a pulling technique is used, each device has to poll every other device on a regular basis. Thus for a network consisting of n devices, n^2 connection are necessary. This results in a high bandwidth and a lot of computing power. The pushing technique bypasses this problem by only sending the state changes when a state change has happened. For the same network consisting of n devices, the best case only uses n connections while the worst case (all applications having a state-change simultaneously) has n^2 connections. The pushing technique thus uses fewer or an equal amount of connections than the pulling technique.

To detect state-changes, the polling technique is used. The polling technique does not need to poll very often in this use case. This is because IoT devices typically only record the environment in the order of seconds. Because of this, the polling can be done in the order of milliseconds and still be accurate. The event-based technique would require altering the source-code of JavaScript.

To prevent synchronization collisions, a timestamp is added to every state-change. This approach assumes in its current form that all clocks are synchronized over the devices. Due to clock drift [31], the clocks of the different devices will differ after some amount of time. There are fortunately a number of ways to synchronize clocks in distributed systems [32, 33]. It is also possible to create a dedicated time server to synchronize across all devices. This time server first fetches the time of all devices in the network, averages these times and reports back to all devices how much they should alternate their time. This is referred as the Berkeley algorithm for clock synchronization and has proven itself useful and efficient [34].

Chapter 5 depicted the time to clone and the amount of synchronization messages between certain numbers of synchronized devices in tables 5.3 and 5.4. The time it took to clone an application without any resources was on average 0.494 seconds. This is about 0.2 seconds more than the time it takes to migrate or fork. This is because the application has to request a syncID from the RR. The same amount of extra time can be found when resources are included. This is because cloning and migration initializations have a lot of shared code. When 2 devices were synchronized,

21 synchronization messages were measured in a time span of 10 seconds. This is 1 more synchronization message more than expected. The target device when hosting a new cloned application first sets its new applications state to completely empty, which is then immediately changed by the initialization code of the application. This results in an extra synchronization message per cloned application. When 3 devices were synchronized only 30 synchronization messages were measured, which is 2 less than expected. This can be caused by a lag in the communication channel. The amount of synchronization messages is linear with the amount of synchronized devices.

6.4 Alternative method for cloning

The current method for synchronizing cloned applications uses a syncID to determine which application is synchronized with which. An alternative method for synchronizing could save all synchronized applications in the application's storage. This would require extra storage but fewer connections would be necessary and no external server would need to be present.

A challenge in this approach is to keep all applications updated with what they need to be synchronized with. When a device running a synchronized application is offline for a short amount of time and then a new application gets synchronized, the application that was offline would not know of this new application. These issues have already been solved by decentralized storage systems [35].

The major advantage of using this approach over the implemented synchronization approach is that no external server is needed, thus no single-point of failure is present in the system. A minor disadvantage of using this approach is that more storage is needed on the IoT devices. This is becoming more irrelevant because the cost associated with SSD storage has been plummeting every year [36]. The storage needed to save data about IoT devices is also tiny. Assuming that every device takes up 100 bytes of memory, 1000 synchronized devices would only take up 100kB of extra storage.

6.5 UI in liquid software

Future work in implementing liquid software in the IoT framework could be the including of UI adaptation in the implemented liquid functionalities. Right now, the migration, forking and cloning do not consider UI and only operate on the

scripting level. Both LfD and LfP consider a different approach to include the UI and adapt it to different devices.

LfD is made for shadow DOM-trees and can thus include libraries like Bootstrap that use CSS-class to make the UI responsive [37]. This also exists for the Polymer project, but to a lesser extent [38]. The choice of technology for the IoT UI is thus an arbitrary one, as long that the technology is sufficiently mature.

What does matter is the way that the UI gets transferred to the new device. One way is to use to subtract the original UI from the current UI to get difference between the UIs or the delta. This is the way LfD works and it does this by the use of virtual DOM-trees. This delta can then be added up to the original UI in the new device to replicate the UI from the source device. Another way is to set components of the UI to liquid and then always send these elements to the new device. This approach is used by LfP. Another way is to send the entire UI to the new device, this would minimize complexity but would increase bandwidth.

7. CONCLUSION

IoT devices have grown in power over the years and will continue to follow this trend. IoT devices will soon be able to host applications. To mass-deploy and manage these IoT-applications, TUT has developed a IoT-framework. Besides this IoT framework, liquid software is on the rise caused by the increasing number of devices per capita. Liquid software states that applications, state and data should not be bound to one device, but should flow between all devices available to the user. There are four main use cases regarding liquid software: migration, forking, cloning and forwarding.

The goal of this Master's thesis was to combine the TUT developed IoT-framework with three of the liquid use cases (migration, forking and cloning). This means that applications hosted on the IoT-devices in the framework should be able to migrate and fork, as well as synchronize their data when cloned.

To migrate and fork applications, the current state of the data is polled and saved to a file. This file is then packed together with the application into a tarball that then gets sent to the target device(s). When a migrate happens, the current application is deleted from the source device. This does not happen when forking. Cloning includes forking an application and keeping both the source and target application synchronized.

To synchronize applications, the state of the application that needs to be synchronized is polled on a regular interval. When the polling detects a change in state, it will send this change to the target application. This can be done in a P2P or master-slave fashion. The synchronization can result in synchronization collisions where two applications end up with different states. To solve this, timestamps were added to the synchronization updates.

To test the liquid functionalities of the IoT framework, a setup was made consisting of four Raspberry Pi 3 Model B's, the RR and IDE running on a virtual machine all contained in a LAN. The RR was connected to an external database. The migration, forking and cloning of an application were all much faster without resources

attached in the source application. When a resources folder is present in the source application, the size of it plays a marginal role. Cloning takes 0.2 seconds longer than migration or forking due to the extra part of receiving a syncID from the RR. The amount of synchronization messages between cloned applications was as much as expected.

The combination of liquid software and the IoT framework resulted in a new tab being created in the IDE. In this tab, N applications could be selected on the left-hand side to do a liquid transfer on M devices selected on the right-hand side of the tab. Three buttons were added for the three liquid functionalities implemented in the framework.

In future work, computational load can be taken off the IoT devices when migrating or forking by using alternative methods. One way is to pack the application at the IDE or RR for migrations and forks with a state file received from the source device. This lowers the computational power needed on the source device but requires all applications to be on the IDE or RR. For cloning, an alternative method can be used as well. All applications could save a list with all other applications it is synchronized with. This would remove the single point of failure that is now present when cloning and would require less connections. It would however require more storage, but this is a minor issue as SSD storage gets more compact and cheaper over the years.

BIBLIOGRAPHY

[1] X. Feng, Y. Laurence, W. Lizhe, and V. Alexey, "Internet of things," *International Journal of Communication Systems*, vol. 25, pp. 1101–1102, 2012.

[2] Statista. (2017) Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions). [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[3] Forbes. (2017) 2017 roundup of internet of things forecasts. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/#5a5bc4fd1480

[4] F. Ahmadighohandizi and K. Systä, "Application development and deployment for iot devices," *Advances in Service-Oriented and Cloud Computing*, pp. 74–85, 2018.

[5] A. Taivalsaari, T. Mikkonen, and K. Syst, "Liquid software manifesto: The era of multiple device ownership and its implications for software architecture," *2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 338–343, July 2014.

[6] J. Hartman, U. Manber, L. L. Peterson, and T. Proebsting, "Liquid software: A new paradigm for networked systems," Tucson, AZ, USA, Tech. Rep., 1996.

[7] A. Gallidabino, C. Pautasso, T. Mikkonen, K. Systä, J.-P. Voutilainen, and A. Taivalsaari, "Architecting liquid software," *Journal of Web Engineering*, vol. 16, pp. 433–470, 2017.

[8] Apple. (2018) Use continuity to connect your mac, iphone, ipad, ipod touch, and apple watch. [Online]. Available: https://support.apple.com/en-us/HT204681

[9] Google. (2018) Create persuasive documents. [Online]. Available: https://www.google.com/intl/en/docs/about/

[10] S. Thangavel and K. Systä, "Liquid transfer of user identity," *Current Trends in Web Engineering*, pp. 92–107, 2018.

[11] Z. P. A, "Accuracy of iphone locations: A comparison of assisted gps, wifi and cellular positioning," *Transactions in GIS*, vol. 13, no. s1, pp. 5–25. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9671.2009.01152.x

[12] w3schools. (2017) Javascript html dom. [Online]. Available: https://www.w3schools.com/js/js_htmldom.asp

[13] J.-P. Voutilainen, T. Mikkonen, and K. Systä, "Synchronizing application state using virtual dom trees," *Current Trends in Web Engineering*, pp. 142–154, 2016.

[14] T. Mikkonen, K. Systä, and C. Pautasso, "Towards liquid web applications," *Engineering the Web in the Big Data Era*, pp. 134–143, 2015.

[15] A. Gallidabino and C. Pautasso, "Maturity model for liquid web architectures," vol. 10360, pp. 206–224, June 2017.

[16] Oracle. (2017) Mysql. [Online]. Available: https://www.mysql.com/

[17] MongoDB. (2017) Nosql databases explained. [Online]. Available: https://www.mongodb.com/nosql-explained

[18] A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Syst, J. P. Voutilainen, and A. Taivalsaari, "On the architecture of liquid software: Technology alternatives and design space," *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 122–127, April 2016.

[19] Webcomponents.org. (2017) Introduction. [Online]. Available: https://www.webcomponents.org/introduction

[20] F. Ahmadighohandizi and K. Systä, "Application development and deployment for iot devices," *Advances in Service-Oriented and Cloud Computing*, pp. 74–85, 2018.

[21] K. Systä and F. Ahmadighohandizi, "Multi-device application development and management through resource discovery for iot," 2017.

[22] NodeJS. (2018) Nodejs. [Online]. Available: https://nodejs.org/en/

[23] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, Mar 2015.

[24] T. Conversations. (2014, January) Software development models: Iterative and incremental development. [Online]. Available: https://technologyconversations.com/2014/01/21/software-development-models-iterative-and-incremental-development/

[25] P. Offermann, S. Blom, M. Schönherr, and U. Bub, "Artifact types in information systems design science – a literature review," R. Winter, J. L. Zhao, and S. Aier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 77–92.

[26] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=2017212.2017217

[27] npm. (2017, December) tar-pack. [Online]. Available: https://www.npmjs.com/package/tar-pack

[28] Lodash. (2018) Lodash: A modern javascript utility library delivering modularity, performance & extras. [Online]. Available: https://lodash.com/

[29] C.-C. Lai and C.-M. Liu, "Approaches for data synchronization on mobile peer-to-peer networks," *Advances in Intelligent Systems and Applications - Volume 2*, pp. 599–608, 2013.

[30] R. Balani, "Energy consumption analysis for bluetooth, wifi and cellular networks," *University of California at Los Angeles*, 2007.

[31] R. Tjoa, K. L. Chee, P. K. Sivaprasad, S. V. Rao, and J. G. Lim, "Clock drift reduction for relative time slot tdma-based sensor networks," in *2004 IEEE 15th International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE Cat. No.04TH8754)*, vol. 2, Sept 2004, pp. 1042–1047 Vol.2.

[32] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 933–940, Aug 1987.

[33] J. van Greunen and J. Rabaey, "Lightweight time synchronization for sensor networks," in *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, ser. WSNA '03. New York, NY, USA: ACM, 2003, pp. 11–19. [Online]. Available: http://doi.acm.org/10.1145/941350.941353

[34] R. Gusella and S. Zatti, "The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 847–853, Jul 1989.

[35] A. Lakshman and P. Malik, "Cassandra - a decentralized structured storage system," April 2010.

[36] Forbes. (2016) The costs of storage. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2016/07/24/the-costs-of-storage/#64d9ccb83239

[37] Bootstrap. (2018) Bootstrap. [Online]. Available: https://getbootstrap.com/

[38] PolymerElements. (2018) app-layout. [Online]. Available: https://github.com/PolymerElements/app-layout

A. TEST MEASUREMENTS

Migrate or Fork			**Clone**		
Resources (Mb)	Tijd (s)	#Target Devices	Resources (Mb)	Tijd (s)	#Target Devices
0	0,284	1	0	0,492	1
0	0,283	1	0	0,512	1
0	0,372	1	0	0,476	1
0	0,296	1	0	0,506	1
0	0,329	1	0	0,485	1
1,9	2,251	1	1,9	1,68	1
1,9	1,286	1	1,9	2,017	1
1,9	1,521	1	1,9	1,543	1
1,9	1,85	1	1,9	2,266	1
1,9	1,079	1	1,9	1,499	1
3,6	1,588	1	3,6	1,681	1
3,6	1,83	1	3,6	6,302	1
3,6	1,489	1	3,6	1,897	1
3,6	1,509	1	3,6	1,689	1
3,6	4,693	1	3,6	1,945	1
0	0,304	2			
0	0,271	2			
0	0,296	2			
0	0,315	2			
0	0,346	2			
0	0,323	3			
0	0,337	3			
0	0,283	3			
0	0,29	3			
0	0,348	3			

Table A.1 *Raw test measurements.*